David Curry
ID: 304755606
CEE/MAE M20
June 15, 2018

Final Project

1 Fluid Behavior

1.1 Introduction

The purpose of this project is to analyze the fluid behavior of smooth particle hydrodynamics on a spatially hashed domain. This is a very broad purpose, as there are many fluid behaviors that I can analyze. I chose to analyze the "dam-break" scenario and a water balloon scenario. The fluid behavior is governed by a few relatively simple forces, but for a system of particles in a fluid, the calculations can become very long and tedious. This script does all the velocity, density, force, and position calculations for every timestep and simulates the results in an iMovie.

1.2 Models and Methods

The first thing the script does is define all the initial fluid conditions and boundary conditions. These conditions are all significant, but they are very repetitive so I only included a few below.

```
dt = 0.003;
tf = 1.5;
xmax = 4;
h = 0.15;
Nx = floor(xmax/h);
dx = xmax/Nx;
```

Next, I created the two structures necessary for the data, particles and bins. This is done using the `struct` function. The numbers for the dam-break scenario are shown below. The water balloon scenario had [20,0] for velocity instead of [0,0].

```
particles(1:n) = struct('pos',[0,0],'vel',[0,0], 'force', [0,0], 'rho',
[], 'neigh',[]);
bins(1:Nx*Ny) = struct('partIDs',[],'adjBins',[]);
```

Next, I included the conditions for both the "dam-break" scenario and the water balloon scenario This included the generation of the initial positions of the particles, which I did with a `rand` function inside a `for` loop of all the particles. I would comment one of these out to analyze the other scenario. This is all shown below.

```
%dam-break
xa = xmax/2;
xb = 0;
ya = ymax;
yb = 0;
for k2 = 1:n
      particles(k2).pos = [(xa-xb)*rand + xb,(ya-yb)*rand+yb];
end
%water balloon
xa1 = 0.5;   xa2 = 0.5;   xa3 = 0.5;
xb1 = 0;     xb2 = 0;     xb3 = 0;
ya1 = 3;     ya2 = 2;     ya3 = 4;
yb1 = 3.3;   yb2 = 2.3;   yb3 = 3.7;
for k2 = 1:n
  if k2 <= n/3
    particles(k2).pos = [(xa1-xb1)*rand + xb1,(ya1-yb1)*rand+yb1];
  elseif k2 > 2*n/3
    particles(k2).vel = [30,5];
    particles(k2).pos = [(xa2-xb2)*rand + xb2,(ya2-yb2)*rand+yb2];
  else
    particles(k2).vel = [8,-10];
    particles(k2).pos = [(xa3-xb3)*rand + xb3,(ya3-yb3)*rand+yb3];
  end
end
```

I then included the code necessary to save the iMovie file with the files conditions. If I was just running the code to check if it worked, I would set `savevid` to false so that the file would not save. This is all shown below.

```
Figure;
savevid = true;
if savevid == true
      vid = VideoWriter('Water4', 'MPEG-4');
      vid.FrameRate = 30;
      vid.Quality = 80;
      open(vid)
end
```

I then created some vectors and a matrix full of zeros to be used later in the code. This is not significant, it just makes the code run a little faster, so it is not shown below. After that, I found the adjacent bins to every bin. This does not change throughout, so this is done outside the timestep loop. A lot of the adjacent bin assignments are repetitive so I only included a few below.

```
for k3 = 1: Nx*Ny
if k3 == 1 %top left corner
  bins(k3).adjBins = [(k3 + 1), (k3 + Ny), (k3 +Ny +1)];
elseif k3 < Ny  %left side
  bins(k3).adjBins = [(k3-1), (k3+1), (k3+Ny-1), (k3+Ny), (k3+Ny+1)];
```

```
      elseif k3 == Ny  %bottom left corner
        bins(k3).adjBins = [(k3 - 1), (k3 + Ny), (k3 + Ny - 1)];
```

Finally, I started the timestep loop that contains all the nested for loops to update the position, velocity, density, force, and neighbors for every particle at every timestep. After creating the timestep loop, I zeroed the A matrix, the particle IDs field of bins and the neighbor particles field of particles. This is repetitive, so just one of each type is shown below.

```
      A = zeros(Ny,Nx);
      for z = 1:Ny*Nx
        bins(z).partIDs = [];
      end
```

Next in the timestep loop, I placed the particles in their correct bins based on their positions. This was done using the bin number formula from Homework 8 and 2 nested `for` loops. This is shown below.

```
      for k5 = 1:Ny*Nx
        for k4 = 1:n
           binNum(k4) = (ceil(particles(k4).pos(1)/dx) - 1)*Ny + ceil((ymax - 
particles(k4).pos(2))/dy);
           if binNum(k4) == k5
               bins(k5).partIDs = [bins(k5).partIDs,k4];
           end
        end
     `end
```

Nest I found the neighbors of every particle and assigned all of the neighbors to the neighbor field of particles. This was done using four nested `for` loops to loop thru the bins, adjacent bins, particles in the current bin, and particles in the adjacent bins. This is all shown below.

```
for z = 1:Nx*Ny
   partBinz = bins(z).partIDs;
   if partBinz ~= 0
      adjacent = (bins(z).adjBins);
      for w = [z,adjacent]
         partBinw = bins(w).partIDs;
         for k6 = partBinz
            for j = partBinw
               dist = sqrt(((particles(k6).pos(1)) - (particles(j).pos(1)))^2 + 
((particles(k6).pos(2)) - (particles(j).pos(2)))^2);
               if (dist < h) && (k6 ~= j)
                  particles(k6).neigh = [(particles(k6).neigh), j];
               end
            end
         end
      end
   end
end
```

I then found the densities of all the particles based on their positions and their neighbors. This was done with two nested for loops: one to loop thru the particles, and one to loop thru the neighbors. This is all shown below.

```
for k7 = 1:n  %loop thru particles
   neighPart = particles(k7).neigh;  %find neighbor particles
   sum = 0;
   for w2 = neighPart  %loop thru neighbors
      norm = sqrt(((particles(k7).pos(1))-(particles(w2).pos(1)))^2 +
((particles(k7).pos(2))-(particles(w2).pos(2)))^2);
      sum = sum + ((4*m)/(pi*h^8))*((h^2 - norm^2)^3);  %sum of contributions
from neighbors
   end
   particles(k7).rho = ((4*m)/(pi*h^2)) + sum; %density formula
end
```

Next, I calculated the force on all the particles based on the densities. This was also done with two nested `for` loops. The sum of the forces was an addition of gravity, pressure forces between particles, and viscous forces between particles. I had to add a constant term *a* to the gravity force because the particles were not falling fast enough in the small time period of my iMovies. This shouldn't change the interactions between the other particles, it will just cause the particles to drop faster. The long force equation is exactly the equation given, it just looks extra long because of the structures title length. This is all shown below.

```
for k8 = 1:n
  f_ext = [0,-9.8]*a*(particles(k8).rho);
  P_k = k*(particles(k8).rho - rho0);
  f = [0,0];
  neighpart2 = particles(k8).neigh;
  for w3 = neighpart2
    P_j = k*(particles(w3).rho - rho0);
    q = (sqrt((particles(k8).pos(1)-particles(w3).pos(1))^2 +
(particles(k8).pos(2) - particles(w3).pos(2))^2)/h);
    f_t = ((m/(pi*h^4*particles(w3).rho))*(1 - q))*((15*k*(particles(k8).rho +
particles(w3).rho - 2*rho0)*((1-q)/q)*(particles(k8).pos -
particles(w3).pos))-(40*mu*(particles(k8).vel - particles(w3).vel)));
    f = f + f_t;
  end
    particles(k8).force = f + f_ext;
end
```

Finally, I updated the particle's position and velocity based on the forces. I also created boundaries for the particles to bounce off. If a particle passed one of the boundaries, it would reverse velocity and go back the other direction. It is important to note that only the perpendicular velocities got reversed. A particle hitting the far right wall only needs to reverse its x velocity to get back into the box. This is all done below in the nested `for` loops.

```
for k9 = 1:n
```

```
        particles(k9).vel = particles(k9).vel + (dt*(particles(k9).force))/
        particles(k9).rho;
        particles(k9).pos = particles(k9).pos + dt*(particles(k9).vel);
        if particles(k9).pos(1) >= xmax
            particles(k9).pos(1) = (2*xmax) - particles(k9).pos(1);
            particles(k9).vel(1) = (-B)*(particles(k9).vel(1));
        elseif particles(k9).pos(1) <= 0
            particles(k9).pos(1) =  0 - particles(k9).pos(1);
            particles(k9).vel(1) = (-B)*(particles(k9).vel(1));
        end
        if particles(k9).pos(2) >= ymax
            particles(k9).pos(2) = (2*ymax) - particles(k9).pos(2);
            particles(k9).vel(2) = (-B)*(particles(k9).vel(2));
        elseif particles(k9).pos(2) <= 0
            particles(k9).pos(2) =  0 - particles(k9).pos(2);
            particles(k9).vel(2) = (-B)*(particles(k9).vel(2));
        end
end
```

Lastly, I plotted the points every timestep to get the iMovie. I used the `plot`, `xlim`, `ylim`, and `grid on` function calls to accomplish this. After the points were plotted I used `writeVideo` to write the iMovie. This is shown below.

```
        for s = 1:n
          x(s) = particles(s).pos(1);
          y(s) = particles(s).pos(2);
          plot(x,y,'bo');
        end
        xlim([0 xmax]);
        ylim([0 ymax]);
        grid on
        if savevid == true
          writeVideo(vid,getframe(gcf))
        else
          drawnow
        end
```

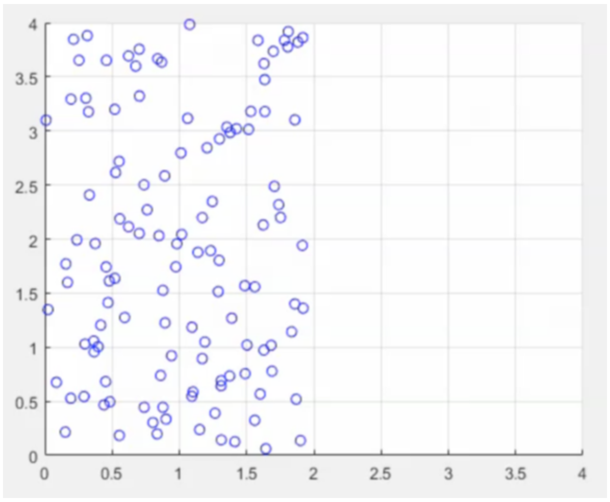
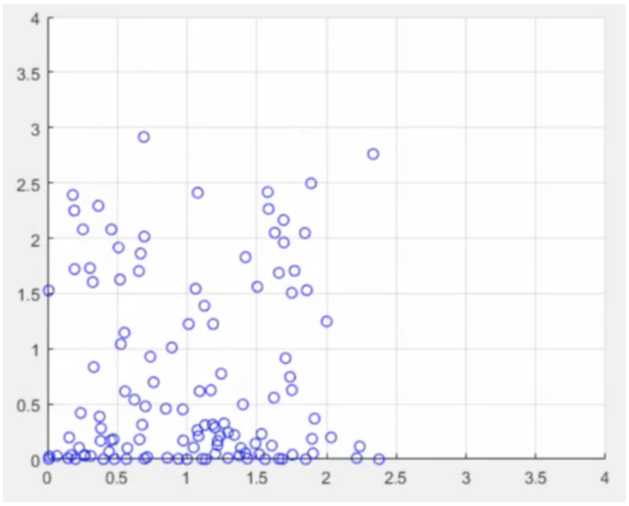
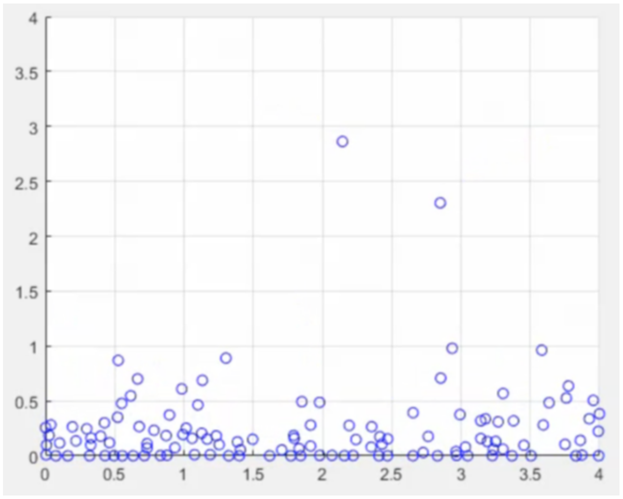
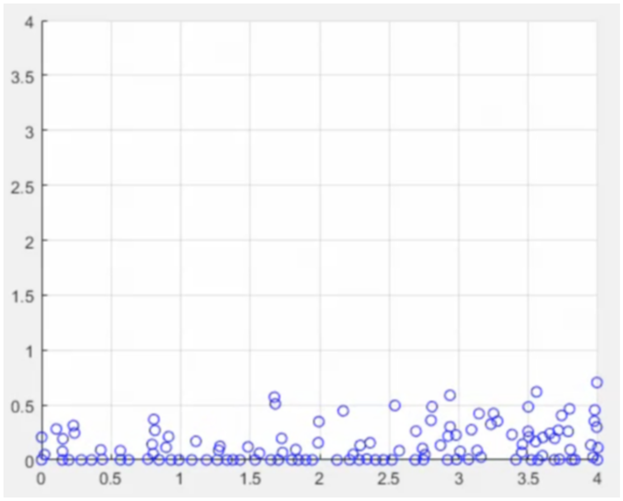
After all of the for loops above, I finally closed the time step loop and closed the video file with the `close(vid)` call.

1.3 Calculations and Results

For the dam break scenario, I used the following inputs and commented out the water balloon initial positions explained above.

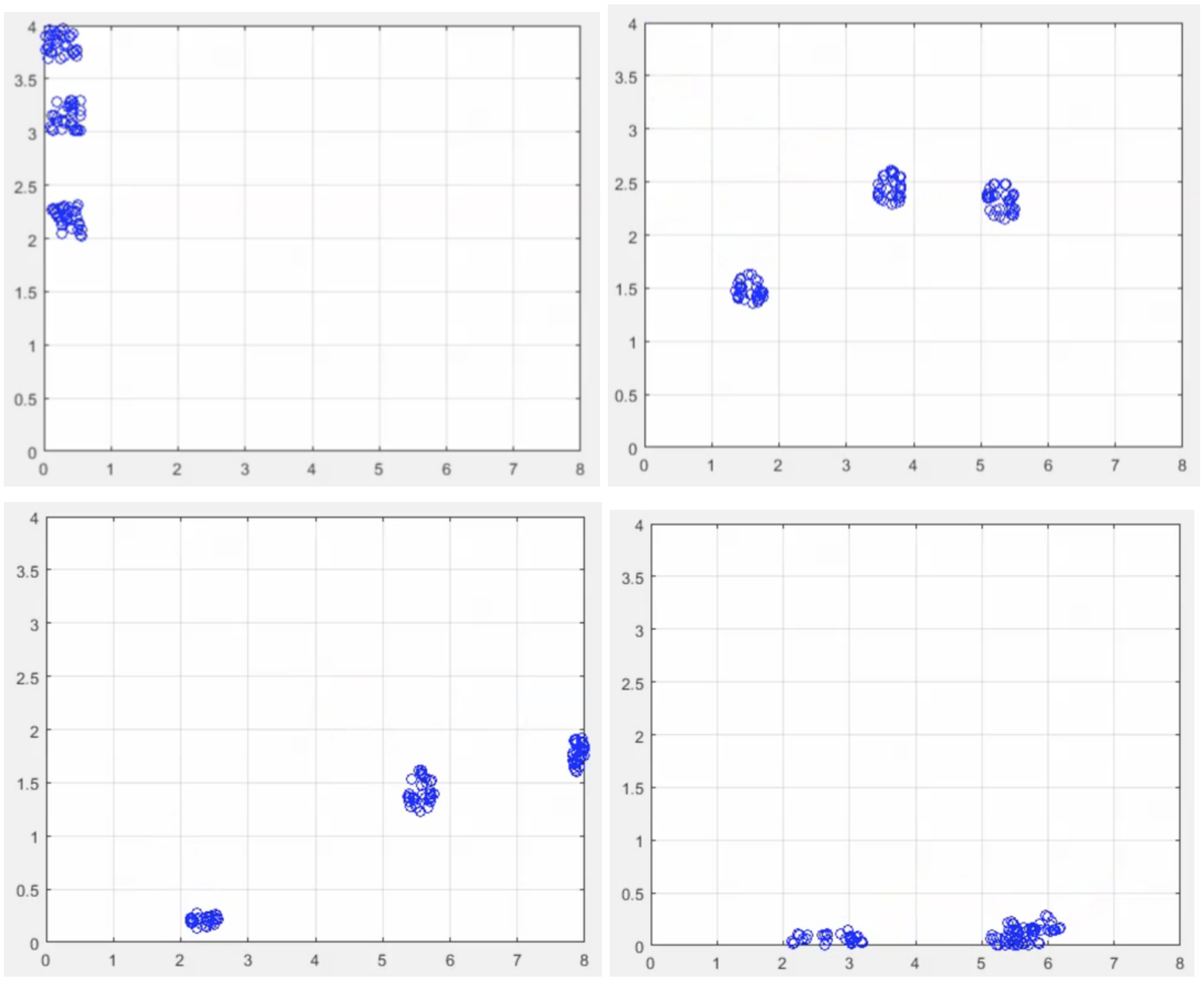
```
dt = 0.003;    k = 80;
tf = 1.5;      mu = 0.8;
B = 0.40;      a = 5;
rho0 = 1500;   xmax = 4;
n = 120;       ymax = 4;
m = rho0/n;    h = 0.1;
```

The output of this script is attached in an iMovie file and a few still images are shown below. The image order is from left to right, then down a row.

For the water balloon scenario, I used the following inputs and commented out the dam break code outlined above. The iMovie file shows the data better, but a few still images are shown below the inputs.

```
dt = 0.003;     k =20;
tf = 0.7;       mu = rho0/n;
B = 0.40;       a = 5;
rho0 = 1500;    xmax = 8;
n = 80;         ymax = 4;
m = rho0/n;     h = 0.5;
```



1.4 Discussion

As shown above in the figures, both scenarios accurately simulated real life conditions. For the dam-break case, the particles clearly fall under gravity and push each other apart into the empty space on the right. The particles also splashed up a few times before settling down, like a fluid actually would. For the water balloon case, the particles all stayed together in a balloon like shape even after they hit a boundary. They deformed, but did not completely lose shape, so this would be represented in real life as if someone did not toss the balloon hard enough to break the plastic, and so they deformed and then bounced back. In the dam-break case I used a large stiffness constant, *k* and a large viscosity whereas in the water balloon case, I used a smaller value for the stiffness and a similar value for the viscosity. It is easier to see in the iMovies, but in the dam-break situation the particles do not stick together that well and bounce off each other more. This is because of the large stiffness constant, and I did this to simulate water droplets bouncing on a low friction surface. For the water balloon case I used lower values for the stiffness so that the particles would clump together into a balloon shape. For both cases I changed the viscosity by about 0.8 in each direction and the lower viscosity particles just bounced a bit more off the walls and off each other. Viscosity is a measure of have much the particles like to stick together, so this makes sense. I also tried mixing two fluids of different densities and the simulation showed the larger density particles clumping together more. It did not really affect the overall movement of the dam-break particles too much.

If I were to create a circular domain, the hashing scheme and the boundary conditions would have to be changed significantly. For the bin assignment scheme I would split the circle up into wedges, and then add several smaller circles inside the big circle to split the bins even more. Assuming the fluid was inside a circle centered at the origin, I set the boundary condition equal to the radius of the circle, or the square root of x squared plus y squared. If a particle ever crossed this boundary, the script could loop thru all the boundary particles to find the nearest boundary particle to the particle crossing it. This can be done by comparing the distance between two boundary particles and setting the smaller of the two to a variable. It would then compare that value to every other distance, updating it if smaller. The script could then update the position of the particle by using the same formulas for the square, and the newly found boundary condition. The velocity would be calculated using the same formula as well.

I also attempted to use the contourf function to make the simulation look like water actually splashing and attempted to add diagonal boundaries, but unfortunately could not get either to work. Those attempts are commented because I ran out of time to try and figure them out.